ОРДЕНА ЛЕНИНА
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
им. М.В.Келдыша

Т.П.Баранова, В.Ю.Вершубский

Использование библиотеки классов пакета “SAGE” для анализа программ, написанных на языке ФОРТРАН

Москва
2004

Т.П.Баранова, В.Ю.Вершубский

Использование библиотеки классов пакета “SAGE” для анализа программ, написанных на языке ФОРТРАН♦

Аннотация

В данной работе представлен опыт использования библиотеки классов Си++ пакета SAGE для анализа больших последовательных программ, написанных на языке ФОРТРАН 77, с целью превращения их в параллельные программы. Описаны структуры данных, используемые пакетом SAGE для внутреннего представления программы. Описана адаптация пакета к использованию его в интерактивной среде с графическим интерфейсом ОС Windows. Кратко описан анализатор текста ФОРТРАН-программ созданный авторами.

T.P.Baranova, V.Yu.Vershoubskii

Usage of the SAGE class library for analyses of FORTRAN programs

Abstract

An experience of usage of the SAGE C++ class library for analyses of large programs written in the FORTRAN 77 language is delivered in this work. The aim is to assist in transformation of the sequential programs into parallel ones. The data structures used by the SAGE package for internal representation of the program to analyze are presented. The adaptation of the package to make it operative in the interactive media with a graphic interface of OS Windows is described. The analyzer of the programs written in the FORTRAN 77 language implemented by the authors is shortly described.

♦ Работа выполнена при финансовой поддержке программы Отделения математических наук РАН (проект 3.2)

Содержание

1. Введение

Развитие вычислительной техники за последнее десятилетие привело к появлению довольно большого числа многопроцессорных вычислительных установок. Доступность этих установок для пользователей, в свою очередь, породила интерес к преобразованию созданных за прошлые годы больших программ, созданных для последовательных машин, в эффективные “параллельные” программы, пригодные для работы на многопроцессорных системах [1 - 4]. Как правило, такое распараллеливание пока производится “вручную”, и в случае успеха приводит к значительному увеличению скорости решения вычислительных задач.

Автоматическое преобразование “последовательных” программ в “параллельные”, несмотря на свою привлекательность, в настоящее время, кажется невыполнимым.

Среди наиболее важных причин здесь можно отметить проблему определения зависимостей, которую существующие методы статического либо динамического анализа программ не позволяют в общем случае решить точно. С другой стороны, неформальные знания о программе часто оказываются, с точки зрения построения эффективной параллельной программы, важнее знаний о наличии или отсутствии зависимости. Эти знания нельзя извлечь автоматически из программных конструкций (например, циклов), так как эти знания выражаются в содержательных терминах, отражают более “крупные” понятия, чем локальные программные конструкции. Например, в вычислительных задачах общая схема эффективного распараллеливания часто определяется характером структур данных, используемых в методе решения (структурные сетки, неструктурные сетки, адаптивные сетки) и общей схемой расчета на этих структурах. Априорные знания о характере структур данных и схеме расчета позволяют обеспечить, например, поддержку работы с распределенным графом на параллельной системе с распределенной памятью, за счет написания вручную соответствующих программ поддержки. Автоматический анализатор, обнаружив в программе массивы с косвенной индексацией, возникающей в данном случае, не сможет осуществить преобразование последовательной программы в эффективную параллельную программу.

Однако, средства анализа последовательных программ, помогающие человеку разобраться в структуре программы, и облегчающие ее преобразование, могут оказаться весьма полезными. Глубина анализа может возрастать постепенно по мере накопления опыта работы с такими средствами.

В данном препринте излагаются первые результаты разработки системы статического анализа последовательных программ, написанных на языке ФОРТРАН 77. Основное назначение системы – автоматизация рутинной работы человека по анализу большой программы с целью ее распараллеливания: определение различных типов зависимостей в программе, прослеживание определения, переопределения, и использования значений переменных, удобная визуализация результатов анализа

и т.п. Таким образом, система обеспечивает человека, проводящего распараллеливание, необходимой информацией.

На первом этапе исследований представлялось интересным выяснить, насколько пригоден для этих целей пакет программ SAGE [5]. Этот пакет был создан в США совместными усилиями сотрудников нескольких университетов и предназначался для анализа и модификации программ, написанных на языках ФОРТРАН и Си. В частности, сами его создатели использовали этот пакет для разработки транслятора с параллельной версии языка программирования Си++.

Препринт состоит из двух частей. В первой части описаны основные структуры данных, которыми пользуется при своей работе пакет SAGE. Сведения собраны на основе экспериментов с пакетом и изучения текстов компонент SAGE. Во второй части описана адаптация пакета к интерактивному режиму работы с графическим интерфейсом и использование пакета для анализа больших программ, написанных на языке ФОРТРАН 77.

Пакет SAGE состоит из парсера, осуществляющего грамматический разбор исходного текста программы и перевод его в машинно-независимое дерево разбора, и библиотеки классов Си++, обеспечивающих работу с этим деревом.

Нами этот пакет использовался пока только для анализа программ (хотя он содержит средства для перестройки программ). В своей работе мы опирались на версию SAGE 1.9 от 05.1995. Пакет распространяется бесплатно. На ftp-сервере [5] находится архивный файл pc++sage++2.0.tar.Z.

2. Краткое описание распространяемого пакета

В упомянутом выше архиве собраны следующие продукты:

- исходные тексты транслятора с языка Си++, расширенного средствами параллелизма,
- исходные тексты программ грамматического разбора текстов на языках ФОРТРАН и Си (парсеры),
- исходные тексты библиотек SAGE - классов,
- демонстрационные программы,
- публикации по данной тематике,
- файлы заданий на языке shell, предназначенные для получения исполняемой программ, и библиотеки классов SAGE.

Все эти средства созданы для работы в пакетном режиме в операционной среде UNIX.

Для превращения исходных текстов в исполняемые модули надо выполнить make-файлы, предварительно настроив их на используемую платформу. После этого, используя полученную библиотеку классов SAGE, можно создавать на языке Си++ свои приложения для анализа и модификации программ.

2.1. Парсер

Парсер производит грамматический разбор исходного файла и порождает на выходе дерево разбора, которое помещает в выходной файл. Информация в выходном файле имеет машинно-независимый вид, в результате чего парсер может работать на вычислительной машине одной архитектуры, а программа анализа, использующая библиотеку классов SAGE – на машине другой архитектуры. Выходной файл всегда имеет расширение dep, а его имя обычно совпадает с именем исходного файла. Поэтому ниже эти файлы будут называться “dep-файлы”.

Парсер обрабатывает исходный текст, последовательно рассматривая предложения программы, и для каждого предложения (кроме комментария) строит дерево разбора. Комментарий считается атрибутом следующего за ним предложения, не являющегося комментарием. Узлы дерева разбора (dep-файла), имеют разную структуру.

2.1.1. Структура узлов дерева разбора

Предложения

bif-узел является корнем поддерева, соответствующего каждому предложению. Он представлен структурой следующего вида:

```
struct bf_nd
{
   u_short id;          // идентификатор данного bif-узла
   u_short variant;     // разновидность данного bif-узла
   u_short cp;          // предок этого узла по управлению
   u_short bf_ptr1;     // ссылка на другое предложение
   u_short cmnt_ptr;    // ссылка на комментарий
   u_short symbol;      // ссылка на символ
   u_short ll_ptr1;     // ссылка на выражение
   u_short ll_ptr2;     // ссылка на выражение
   u_short ll_ptr3;     // ссылка на выражение
   u_short dep_ptr1;    // ссылка на зависимости
   u_short dep_ptr2;    // ссылка на зависимости
   u_short label;       // ссылка на метку
   u_short lbl_ptr;     // ссылка на метку
   u_short g_line;      // глобальный номер строки
   u_short l_line;      // локальный номер строки
   u_short decl_specs; // ссылка на спецификатор формата
   u_short filename;    // ссылка на имя исходного файла
};
```

Все поля структуры имеют тип `u_short`, представляющий целое без знака. В зависимости от машины и системы программирования он может иметь разную длину.

Поле `id` это - уникальный для множества структур данного типа целочисленный идентификатор – номер узла. Все ссылки в полях структур dep-файла это – ссылки на идентификаторы других структур.

Поле `variant` это - целочисленная характеристика, которая показывает, какую сущность исходного файла представляет данный узел. Например, в случае исходного текста на языке ФОРТРАН, bif-узел может представлять заголовок программной единицы, предложение присваивания, предложения `GOTO`, `IF`, `DO`, `COMMON`, и т.п.

Поле `g_line` содержит номер начальной строки соответствующего предложения, относительно начала исходного файла.

Поле `l_line` содержит номер строки, в которой начинается данное предложение, относительно заголовка той программной единицы, в которой оно находится. Для самого заголовка значение этого поля равно 1.

Поля `ll_ptr1`, `ll_ptr2`, `ll_ptr3` представляют собою ссылки на выражения, например, на левую и правую части предложения присваивания, на условия предложений `IF`, на выражения, определяющие начальное значение, конечное значение и шаг изменения переменной цикла предложений `DO`, и т.п.

Выражения

Выражения представлены поддеревом, узлам которого соответствуют ll-структуры.

```
struct ll_nd
{
   u_short id;
   u_short variant;
   u_short type;
};
```

В этой структуре отсутствуют ссылки, необходимые для построения дерева выражения, хотя такая информация в dep-файле содержится. Приведенная выше структура отражает лишь “голову” узла поддерева выражения. Поля `id`, `variant`, `type` присутствуют во всех узлах данного типа.

Вслед за полями приведенной выше структуры могут следовать и другие поля, однако их количество и смысл зависит от той сущности (от значения поля `variant`), которую представляет конкретный узел.

Символы

Листья дерева разбора (лексемы) представлены в dep-файле структурой следующего вида:

```
struct sym_nd
{
   u_short id;       // идентификатор (номер) символа
```

```
   u_short variant; // вид символа
   u_short type;    // ссылка на тип данных
   u_short attr;    // ссылка на атрибут
   u_short next;    // ссылка на следующий символ
   u_short scope;   // область видимости
   u_short ident;   // ссылка на символьную строку (имя)
};
```

Листья дерева разбора в системе SAGE носят названия символов.

Типы данных

Типы данных в dep-файле представлены структурами следующего вида.

```
struct typ_nd
{
   u_short id;
   u_short variant;
   u_short name;
};
```

Как и в случае узлов нижнего уровня, эта структура представляет собою лишь “голову” соответствующих объектов, поскольку типы данных (особенно в случае использования языка Си) могут иметь весьма сложную конструкцию.

В области размещения узлов типа в dep-файле всегда присутствуют узлы, представляющие встроенные типы данных, независимо от присутствия их в исходном тексте программы. В случае языка ФОРТРАН 77 это - `INTEGER, REAL, DOUBLE PRECISION, CHARACTER, LOGICAL, COMPLEX, DOUBLE COMPLEX.`

Метки предложений

Метки в dep-файле представлены структурами следующего вида:

```
struct lab_nd
{
   u_short id;      // идентификатор (номер) узла
   u_short labtype; // тип метки
   u_short body;    //
   u_short name;    //
   long    stat_no; //
};
```

Ссылки на файлы

Ссылки на файлы представлены в dep-файле структурами следующего вида:

```
  struct fil_nd
  {
       u_short id;   // идентификатор (номер) узла
       u_short name; // ссылка на имя файла
  };
```

Комментарий

Комментарий в dep-файле представлен структурой следующего вида:

```
struct cmt_nd
{
     u_short id;   // идентификатор (номер) узла
     u_short type; // тип комментария
     u_short next; // ссылка на следующий узел
     u_short str;  // ссылка на символьную строку
};
```

2.1.2. dep-файл

Разные узлы появляются по мере разбора исходного текста, однако в dep-файл узлы укладываются по сортам, образуя свои области.

В приведенных ниже примерах представлены фрагменты dep-файла следующей простой ФОРТРАН-программы:

```
SUBROUTINE ALPHA(FA0,FA1,FA2,FA3)
COMMON /XXX/A0,A1,A2,A3,A4,A5,A6,A7,A8,A9
DOUBLE PRECISION A0,A1,A2,A3,A4,A5,A6,A7,A8,A9
DOUBLE PRECISION X,Y,Z
EQUIVALENCE (FA1, PSI)
FA2=22
A9=(X-Y)*Z
X=FA2
Y=A9
CALL CHARLY(A2,10,A5+1.0,X)
FA3 = 75.0
PSI = X*Y
END
```

Содержимое dep – файла представлено в 16-м формате по 16 байтов в строке. В первой колонке находятся адреса первых байтов соответствующих строк. В конце представлено содержимое строки в текстовом виде, там, где это возможно.

```
00000000:73 61 67 65 2E 64 65 70|10 01 00 00 B8 04 00 00 | sage.dep
00000010:20 0D 00 00 A0 10 00 00|50 11 00 00 50 11 00 00 |
00000020:50 11 00 00 58 11 00 00|58 11 00 00 20 00 00 00 |
00000030:00 00 00 00 0D 00 00 00|0E 00 00 00 5B 00 00 00 |
                        .....
```

Первые 8 байтов dep-файла имеют предопределенное значение. Они служат удостоверением того, что данный файл является легитимным продуктом парсера SAGE-пакета:

```
00000000:73 61 67 65 2E 64 65 70|10 01 00 00 B8 04 00 00 | sage.dep
```

Далее следует константа, также имеющая предопределенное значение. Она используется для того, чтобы понять, как упорядочены байты образующие числовые значения. Сразу же заметим, что в нашем случае (процессор Intel Pentium) байты,

содержащие старшие разряды целочисленных полей, занимают старшие адреса файла, т. е. в приведенном ниже фрагменте жирным шрифтом представлена константа `0x00000110`.

```
00000000:73 61 67 65 2E 64 65 70|10 01 00 00 B8 04 00 00 | sage.dep
```

Далее размещена структура, поля которой указывают на положение в dep-файле областей для размещения узлов разных видов в виде их смещения от начала файла.

```
struct locs
{ // структура, указывающая положение областей .dep-файла
  long llnd;     // смещение области ll-узлов
  long symb;     // смещение области symbol-узлов
  long type;     // смещение области type-узлов
  long labs;     // смещение области label-узлов
  long cmnt;     // смещение области comment-узлов
  long file;     // смещение области filename-узлов
  long deps;     // смещение области dependence-узлов
  long strs;     // смещение области строчных констант
};
```

В приведенном ниже примере затененной областью показано положение структуры `locs` в dep – файле. В этом примере подчеркнутым шрифтом выделено поле `strs`, указывающее на положение области строчных констант файла, а ниже приведена сама область строчных констант.

```
00000000:73 61 67 65 2E 64 65 70|10 01 00 00 B8 04 00 00 | sage.dep
00000010:20 0D 00 00 A0 10 00 00|50 11 00 00 50 11 00 00 |
00000020:50 11 00 00 58 11 00 00|58 11 00 00 20 00 00 00 |
                                .
                                .
                                .
00001150:.. .. .. .. .. .. .. ..|19 00 00 00 03 00 00 00 |
00001160:31 2E 30 04 00 00 00 37|35 2E 30 01 00 00 00 2A | 1.0    75.0    *
00001170:05 00 00 00 61 6C 70 68|61 03 00 00 00 66 61 30 |     alpha    fa0
00001180:03 00 00 00 66 61 31 03|00 00 00 66 61 32 03 00 |     fa1     fa2
00001190:00 00 66 61 33 03 00 00|00 78 78 78 02 00 00 00 |   fa3     xxx
000011A0:61 30 02 00 00 00 61 31|02 00 00 00 61 32 02 00 | a0    a1    a2
000011B0:00 00 61 33 02 00 00 00|61 34 02 00 00 00 61 35 |   a3    a4    a5
000011C0:02 00 00 00 61 36 02 00|00 00 61 37 02 00 00 00 |     a6    a7
000011D0:61 38 02 00 00 00 61 39|01 00 00 00 78 01 00 00 | a8    a9    x
000011E0:00 79 01 00 00 00 7A 03|00 00 00 70 73 69 06 00 |  y    z    psi
000011F0:00 00 63 68 61 72 6C 79|09 00 00 00 41 4C 50 48 |   charly    ALPH
00001200:41 2E 46 4F 52         |                       | A.FOR
```

Область строчных констант располагается в самом конце dep – файла и по своей структуре отличается от остальных областей. В ней размещены все имена переменных, имена программных единиц, строки комментария, и т. п. В начале этой области находится число (выделено подчеркнутым шрифтом), показывающее ко-

личество представленных в ней строк (в данном примере 0x19 = 25). Поскольку строки имеют переменную длину, они представлены в виде пар: <размер> - <содержимое>. Каждая строка представлена числом содержащихся в ней байтов (подчеркнутый шрифт) и собственно байтами, образующими строку (курсив). Для примера затененными областями выборочно показано три строки.

```
00001150:01 00 00 00 18 00 00 00|19 00 00 00 03 00 00 00 |
00001160:31 2E 30 04 00 00 00 37|35 2E 30 01 00 00 00 2A | 1.0    75.0    *
00001160:31 2E 30 04 00 00 00 37|35 2E 30 01 00 00 00 2A | 1.0    75.0    *
000011F0:00 00 63 68 61 72 6C 79|09 00 00 00 41 4C 50 48 |   charly    ALPH
00001200:41 2E 46 4F 52         |                       | A.FOR
```

Такая конструкция области позволяет легко получить из нее в оперативной памяти массив строк стандарта языка Си.

Вслед за структурой, показывающей положение областей в dep-файле, следует структура следующего вида (преамбула):

```
struct preamble
{ // структура преамбулы dep-файла
     u_short ptrsize;    // кол-во бит в указателях(32 или 64)
     u_short language;   // язык исходного текста (ФОРТРАН или Си)
     u_short num_blobs;  // кол-во blob-узлов
     u_short num_bfnds;  // кол-во bif-узлов
     u_short num_llnds;  // кол-во ll-узлов
     u_short num_symbs;  // кол-во symbol-узлов
     u_short num_types;  // кол-во type-узлов
     u_short num_label;  // кол-во label-узлов
     u_short num_dep;    // кол-во dep-узлов
     u_short num_cmnts;  // кол-во comment-узлов
     u_short num_files;  // кол-во filename-узлов
     u_short global_bfnd;// id глобального bif-узла
};
```

В приведенном ниже фрагменте затененной областью показано положение этой структуры в dep – файле:

```
00000000:73 61 67 65 2E 64 65 70|10 01 00 00 B8 04 00 00 | sage.dep
00000010:20 0D 00 00 A0 10 00 00|50 11 00 00 50 11 00 00 |
00000020:50 11 00 00 58 11 00 00|58 11 00 00 20 00 00 00 |
00000030:00 00 00 00 0D 00 00 00|0E 00 00 00 5B 00 00 00 |
00000040:16 00 00 00 0C 00 00 00|00 00 00 00 00 00 00 00 |
00000050:00 00 00 00 01 00 00 00|01 00 00 00 01 00 00 00 |
00000060:64 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00 |
```

В преамбуле указано количество узлов в каждой из областей dep-файла. По поводу blob – узлов следует заметить, что они не образуют в dep – файле отдельную область, а находятся в области bif – узлов. За каждым bif-узлом следуют две blob-последовательности чисел типа `u_short`. В каждой последовательности первое число задает ее длину, а последующие числа – ее элементы. Если последователь-

ность - пустая, то ее длина задается равной 0, а элементов нет. Поэтому за каждым bif-узлом следует, по крайней мере, два числа.

Первым узлом bif-области является так называемый глобальный bif-узел. Он не соответствует никакому предложению исходного текста, а является заголовком dep-файла. Следующая за глобальным bif-узлом blob-последовательность отражает последовательность программных единиц dep-файла. Значения элементов этой последовательности равны значениям полей `id` bif-узлов, представляющих предложения `PROGRAM, SUBROUTINE, FUNCTION`.

За bif-узлом, представляющим заголовок программной единицы, находится blob - последовательность, которая отражает последовательность всех предложений, входящих в состав этой единицы, (элементы blob-последовательности ссылаются на `id` всех предложений программной единицы).

В преамбуле показано суммарное количество blob-узлов, задействованных в dep-файле. Все эти blob - последовательности будут развернуты в blob-структуры при использовании dep-файла.

2.2. Библиотека SAGE - классов

Библиотека реализована в виде иерархии классов. После преобразования парсером исходного текста анализируемой программы, методы классов обеспечивают доступ к дереву разбора. Классы объединены в пять семейств:

- “Проекты и файлы” – содержат средства для работы с файлами.
- “Предложения” – содержат средства для работы с предложениями.
- “Выражения” – содержат средства для работы с выражениями, входящими в состав предложений.
- “Символы” – содержат средства для работы с идентификаторами.
- “Типы” – содержат средства для работы с типами идентификаторов.

Каждое семейство опирается на базовый класс и может содержать некоторое количество наследников. Базовые классы и их семейства кратко описаны ниже:

SgProject. Наследников нет. Класс рассматривает проект как массив dep - файлов. Методы класса позволяют узнать количество файлов, входящих в состав проекта, получить ссылку на любой файл и имя любого файла по индексу массива, выяснить язык программирования исходной программы (Си или ФОРТРАН), добавить файл к проекту и удалить файл из проекта.

SgFile. Наследников нет. Класс рассматривает файл как массив программных единиц. Методы класса позволяют узнать количество программных единиц в файле, получить ссылку на заголовок программной единицы по индексу массива, получить ссылку на главную программу (PROGRAM в ФОРТРАНе или main в Си), узнать количество структур и получить ссылку на структуру по индексу. Методы класса позволяют также получить ссылку на первое предложение, на первое выражение, на первый символ или на первый тип в файле, а также на

предложения, заданные их уникальными идентификаторами. Кроме того, есть методы, позволяющие выдать файл (возможно уже измененный) в машинно-независимой форме (в виде dep-файла) или в виде текста на исходном языке программирования.

SgStatement. 68 наследников. Этот базовый класс и его наследники образуют семейство, которое рассматривает программу как совокупность предложений. Объекты классов семейства соответствуют предложениям исходной (или уже измененной) программы. Методы класса обеспечивают доступ к свойствам этих объектов, таких как разновидность предложения, номер строки в исходном тексте, наличие метки предложения, наличие комментария. Можно получить ссылку на предыдущее или на следующее (в лексическом смысле) предложение и на предка по иерархии, например, на предложение `DO`, `IF` или `SUBROTINE`, в теле которого находится данное предложение. Многим конкретным типам предложений соответствуют классы – наследники. Они предоставляют дополнительные специфические методы, например, для структурного `IF` получить ссылки на первые предложения блока `TRUE` и блока `FALSE`. Для предложения присваивания получить ссылки на выражения, соответствующие левой и правой части присваивания. Для предложения `DO` получить ссылки на выражения, определяющие начальное значение, конечное значение и шаг изменения переменной цикла, а также на символ переменной цикла, и т.п. Есть метод, позволяющий получить фрагмент дерева, соответствующий объекту в текстовом виде.

SgExpression. 52 наследника. Этот базовый класс и его наследники образуют семейство, которое используются для представления выражений, входящих в состав предложений. Объект этого класса соответствует элементу выражения – операции, операнду, элементу списка. Основные свойства объекта это – его разновидность (сложение, вычитание, элемент списка, ссылка на простую переменную, ссылка на переменную с индексами, константа, и т.п.), ссылка на тип данных (целый, действительный, комплексный, массив целых и т.п.) и ссылки на операнды и на индексы. Методы класса обеспечивают доступ к свойствам объекта. Например, для самоопределенной целочисленной константы можно получить ее значение. Для арифметических и логических операций можно получить ссылки на операнды. Для ссылки на простую переменную можно получить ссылку на соответствующий ей символ. Можно получить ссылку на тип выражения, и т.п. Есть метод, позволяющий получить фрагмент дерева, соответствующий выражению в текстовом виде.

SgSymbol. 11 наследников. Этот класс и его наследники образуют семейство, объекты которого соответствуют переменным в программе, именам программных единиц, именам COMMON-блоков, структур, полей структур и т.д. Свойства

объектов отражают имя, тип данных, область локализации переменной и т.п. Методы класса обеспечивают доступ к свойствам объектов.

SgType. 12 наследников. Этот класс и его наследники образуют семейство, объекты которого представляют типы данных. На них ссылаются выражения и символы. Типы бывают простые (`INTEGER, REAL, COMPLEX`) и структурные (массивы, указатели). Методы класса позволяют понять, какой это тип. Для массива можно получить его размерность, базовый тип, список размеров по разным измерениям.

SgAttribute. Нет наследников. Методы этого класса дают пользователю возможность присоединить к объектам классов SgStatement, SgExpression, SgSymbol и SgType дополнительные структуры данных по его усмотрению.

Таким образом, методы классов предоставляют широкий набор средств для перемещения по дереву программы и для анализа компонент дерева. В большинстве классов присутствуют также методы, позволяющие создавать, копировать, изменять и удалять объекты (средства для изменения программы).

2.3. Использование SAGE пакета

Для того чтобы можно было пользоваться библиотекой классов SAGE, надо сначала пропустить через парсер исходный текстовый файл анализируемой программы и получить соответствующий dep - файл. Если программа состоит из нескольких файлов, то упомянутое выше действие надо выполнить для каждого файла.

Затем с помощью текстового редактора надо создать файл проекта, в котором перечислены имена dep-файлов, составляющих полную программу. Каждое имя файла записывается на отдельной строке. Файл проекта должен иметь расширение prj .

После этого можно создать и запустить программу анализа, использующую функции из библиотеки SAGE. Программа анализа должна содержать предложение `#include "sage++user.h"` и первой должна вызываться функция – конструктор класса `SgProject`:

```
SgProject  *pProject = new SgProject(<имя файла проекта>);
```

2.4. Объекты классов SAGE и используемые функции

Дерево разбора, содержащееся в dep-файле, адекватно отражает программу, однако лишь в логическом смысле. Узлы этого дерева не являются объектами классов SAGE. Упомянутый выше конструктор читает dep-файлы проекта и на их основе строит в памяти машины граф анализируемой программы в терминах объектов классов SAGE, а также порождает другие необходимые для работы библиотеки структуры данных. Ниже рассмотрена работа этого конструктора.

```
SgProject  *pProject(char *proj_file_name)
{
     InitVariable();
     open_proj_toolbox(proj_file_name, proj_file_name);
     {
          Открывает файл проекта
          Создает список имен dep-файлов - file_list
          Определяет количество dep-файлов - no
          OpenProj(proj_name, no, file_list);
          {
               Создает объект SgProect
               Создает список проектов
               Создает хэш-таблицу для программных единиц проекта
               open_proj_file(p, no, file_list);
               {
                    Для каждого dep-файла из списка
                    open_file(fp);
                    {
                         Открывает dep-файл
                         Создает для него объект SgFile *f
                         read_nodes(f);
                         {
                              need_swap = 0;       // static
                              lfi = fi;            // static
                              fd = fi->fid;        // static
                              read_preamble();
                              read_bif_nodes();
                              read_ll_nodes();
                              read_symb_nodes();
                              read_type_nodes();
                              read_label_nodes();
                              read_cmnt_nodes();
                              read_filename_nodes();
                              read_dep_nodes();
                         }
                         Закрывает dep-файл
                         Создает хэш-таблицу для символов файла
                         return(f); // Ссылка на SgFile
                    }
                    Включает объект в цепочку файлов проекта
                    Включает программные единицы dep-файла в
                    хэш-механизм проекта
               }
               return (p); // Ссылка на SgProject
          }
     }
     Init_Tool_Box();
     CurrentProject = this;
}
```

Конструктор SgProject вызывает функцию

```
open_proj_toolbox(proj_file_name, proj_file_name)
```

передавая ей имя файла проекта. Эта функция открывает файл проекта, выясняет количество dep-файлов в проекте и строит в памяти список имен dep-файлов – соответствующие текстовые строки и массив указателей на них, - после чего вызывает функцию `OpenProj(proj_name,no,file_list)`, передавая ей имя файла проекта, количество dep-файлов в проекте и список имен dep-файлов.

Функция OpenProj()

Создает объект SgProject, который имеет приведенную ниже структуру, и порождает список проектов, состоящий пока из одного элемента.

```
struct proj_obj {
  char    *proj_name; //имя файла проекта
  PTR_BLOB file_chain;//список dep-файлов проекта
  PTR_BLOB hash_tbl[];//хэш-таблица всех программных единиц
  PTR_PROJ next;      //указатель на следующий проект
};
```

Функция `OpenProj()` подготавливает хэш-таблицу для программных единиц проекта. Таблица пока пустая. После этого происходит вызов функции `open_proj_file(p,no,file_list)`, которой в качестве параметров передаются указатель на проект, количество dep-файлов в проекте и список имен dep-файлов.

Функция open_proj_file()

В цикле открывает упомянутые в списке dep-файлы и создает для каждого файла структуру данных соответствующую объекту класса SgFile, и вызывает функцию `read_nodes()`, передавая ей указатель на эту структуру:

```
struct file_obj {
  char  *filename;       // имя dep файла
  FILE  *fid;            // дескриптор файла
  Int   lang;            // язык программирования
  PTR_HASH *hash_tbl;    // хэш-таблица для доступа к символам
  PTR_BFND  global_bfnd; // ук.на глобальный BIF-узел файла
  PTR_BFND  head_bfnd;   // ук.на первый BIF-узел файла
  PTR_BFND  cur_bfnd;    // ук.на текущий BIF-узел файла
  PTR_LLND  head_llnd;   // ук.на первый LL-узел файла
  PTR_LLND  cur_llnd;    // ук.на текущий LL-узел файла
  PTR_SYMB  head_symb;   // ук.на первый SYMBOL-узел файла
  PTR_SYMB  cur_symb;    // ук.на текущий SYMBOL-узел файла
  PTR_TYPE  head_type;   // ук.на первый TYPE-узел файла
  PTR_TYPE  cur_type;    // ук.на текущий TYPE-узел файла
  PTR_BLOB  head_blob;   // ук.на первый BLOB-узел файла
  PTR_BLOB  cur_blob;    // ук.на текущий BLOB-узел файла
  PTR_DEP   head_dep,    // ук.на первый DEP-узел файла
```

```
  PTR_DEP   cur_dep;      // ук.на текущий DEP-узел файла
  PTR_LABEL head_lab;     // ук.на первый LABEL-узел файла
  PTR_LABEL cur_lab;      // ук.на текущий LABEL-узел файла
  PTR_CMNT  head_cmnt;    // ук.на первый COMNT-узел файла
  PTR_CMNT  cur_cmnt;     // ук.на текущий COMNT-узел файла
  PTR_FNAME head_file;    // ук.на первый файл области файлов
  int    num_blobs;       // кол-во blob узлов
  int    num_bfnds;       // кол-во bif узлов
  int    num_llnds;       // кол-во ll узлов
  int    num_symbs;       // кол-во symb узлов
  int    num_label;       // кол-во label узлов
  int    num_types;       // кол-во type узлов
  int    num_files;       // кол-во filename узлов
  int    num_dep;         // кол-во dependence узлов
  int    num_cmnt;        // кол-во comment узлов
};
```

Функция read_nodes()

Производит импорт данных из dep-файла в структуры данных SAGE-классов. В результате работы функции read_nodes() все поля приведенной выше структуры будут заполнены, кроме поля `hash_tbl`. По выходе из функции `read_nodes()` граф анализируемой программы будет в основном построен.

Для быстрого доступа к символам по имени для каждого объекта SgFile строится хэш-таблица, представляющая собой массив указателей на вспомогательные структуры следующего вида:

```
struct hash_entry  {
   char *ident;                    // ссылка на имя символа
   struct hash_entry *next_entry; // ссылка на следующую стр-ру
   PTR_SYMB id_attr;               // ссылка на символ
};
```

Для доступа к нужному символу используется хэш-функция, которая для заданного имени возвращает индекс этого массива. Поле `next_entry` служит для разрешения хэш-коллизий. Поле `id_attr` содержит ссылку на соответствующий символ, а поле `ident` – ссылку на имя идентификатора.

По выходе из функции `open_file(f)` производится включение очередного объекта класса SgFile в цепочку dep-файлов проекта. Кроме того, хэш-таблица проекта пополняется именами программных единиц, содержащимися в очередном dep-файле. Когда все dep-файлы проекта будут прочитаны, все необходимые для работы с пакетом SAGE структуры будут построены.

Ниже импорт данных из dep-файла в SAGE-структуры рассмотрен более подробно. Функция `read_nodes(f)` копирует некоторые локальные переменные в статические глобальные переменные (для того, чтобы не передавать их вызывае-

мым функциям через параметры) и последовательно вызывает 9 функций для чтения разных областей dep-файла.

Функция read_preamble()

Производится проверка легитимности dep-файла. Первые 8 байтов файла, порожденного парсером, должны содержать последовательность знаков `sage.dep`. Они считываются из файла в буфер и сравниваются с соответствующей текстовой константой. Если результат сравнения отрицательный, то работа прекращается.

Выясняется порядок следования байтов, представляющих в dep-файле числа. (dep-файл мог быть порожден на вычислительной машине другой архитектуры). Число, следующее за упомянутой выше строкой, должно иметь значение – `0x0110`. Это число считывается из файла в буфер и сравнивается с соответствующей константой. Если результат сравнения отрицательный, то порядок следования байтов в буфере изменяется на обратный, после чего сравнение повторяется. Если результат сравнения опять отрицательный, то работа прекращается. В зависимости от того, в каком случае сравнение дало положительный результат, делается вывод о необходимости преобразования полей буферной структуры данных, несущих числовую информацию, после считывания информации из dep-файла (флаг `need_swap`).

В буфер, имеющий тип структуры dep-файла `struct locs` (см. выше), считывается из файла соответствующее количество байтов. Если необходимо, порядок следования байтов в полях этой структуры меняется на обратный. В результате оказываются доступными координаты всех областей dep-файла.

Литеральные строки из последней области dep-файла (имена переменных, имена программных единиц, имена файлов и т.п.) импортируются в SAGE-структуру данных в виде символьных строк языка Си. Указатель dep-файла устанавливается на начало этой области. Считывается первое число типа u_short. Это – количество символьных строк. Создается массив указателей на нужное число строк. Затем в цикле для каждой строки из файла выполняются шаги:

- считывается длина строки,
- выделяется динамическая память, для размещения символов этой строки,
- символы строки считываются из файла в выделенную память,
- строка завершается признаком конца,
- ссылка на выделенную память помещается в массив указателей.

Указатель dep-файла устанавливается на точку `0x2C` относительно начала файла. Здесь размещена информация о размере указателей, о языке программирования и о количестве узлов дерева разбора исходной программы в каждой из областей dep-файла. В переменную типа `struct preamble` из файла считывается необходимое число байтов. В случае необходимости производится преобразование

полей структуры. Теперь известно количество узлов каждого типа, использованных для построения графа программы.

Заполняются поля структуры объекта SgFile, содержащие сведения о языке и о количестве узлов разного типа.

Для множества узлов каждого типа выделяется динамическая память, необходимая для размещения этих узлов. Здесь тип узлов соответствует объекту SAGE-класса (а не структуре dep-файла). Указатели на первый узел каждого типа помещаются в рабочие переменные и в соответствующие поля объекта SgFile. В другие поля объекта SgFile помещаются указатели на последние узлы в каждом участке памяти. В результате создается заготовка графа анализируемой программы в терминах объектов SAGE–классов (получены все необходимые для построения графа узлы, но узлы пока еще пустые).

В рамках каждого типа узлы нумеруются, и для каждого узла его личный номер помещается в поле узла (`id`). Это – уникальный в пределах конкретного типа идентификатор узла. Кроме того, для каждого типа узлы объединяются в односвязный список - в поле `thread` каждого узла помещается ссылка на следующий узел. В поле `thread` последнего узла помещается ссылка `NULL` – признак конца списка. SAGE-структуры готовы к приему информации из dep-файла.

Функция read_bif_nodes()

bif-узел представляет в графе программы предложение. Поскольку разновидностей предложений много (SAGE учитывает, как несколько разновидностей языка ФОРТРАН, так и Си++), то разновидностей bif-узлов тоже много. Это разнообразие учтено в структуре bif-узла тем, что помимо полей, присутствующих в узлах любого типа, имеется объединение из 20 вариантов разных структур. Однако все эти структуры имеют одинаковый размер и одинаковое количество полей – просто в разных структурах используются разные поля. Среди этих вариантов есть один служебный вариант `Template`, в котором упомянуты все поля. Этот вариант объединения и используется для импорта данных из dep-файла.

В dep-файле все bif-узлы представлены одной структурой. Структура bif-узлов dep-файла и структура bif-узлов SAGE-графа совпадают лишь частично. Несмотря на это, импорт данных из bif-узлов dep-файла в bif-узлы SAGE-графа осуществляется сравнительно просто. Количество bif-узлов и их местонахождение в dep-файле известно. Поскольку bif-узлы следуют непосредственно за преамбулой, то после чтения преамбулы dep-файл уже находится в нужной позиции.

В цикле каждый bif-узел dep-файла считывается в переменную соответствующего типа. В случае необходимости поля этой переменной подвергаются преобразованию. Поскольку указатель на первый bif-узел SAGE-графа известен, то и указатель на i-й узел получить легко. Значения некоторых целочисленных полей, таких как разновидность узла (`variant`) или номер строки (`g_line, l_line`), просто копируются. Однако поля, представляющие собою ссылки, копировать нельзя.

Ссылки в dep-файле это - числа (`id` соответствующих узлов), тогда как ссылки в SAGE-графе должны быть адресами. Необходимое преобразование осуществляется макросом `NULL_CHECK()`

```
#define NULL_CHECK(BASE,VALUE) ((VALUE)?(BASE+(VALUE-1)):0)
```

`BASE` это – указатель на первый узел соответствующего множества узлов SAGE-графа. `VALUE` – целочисленная ссылка dep-файла. Если она равна 0, то макрос возвращает в качестве значения адреса `NULL`. Если `VALUE` – ссылка на узел с номером n, то макрос возвратит адрес n-го узла SAGE-графа.

Вслед за каждым bif-узлом в dep-файле следует две blob-последовательности, быть может, пустых. Элементы последовательности – целые числа. Первое число в каждом списке это – его длина (для пустого списка равная 0). Функция `read_blob_nodes()` трансформирует эту последовательность в SAGE-список blob-узлов. Указатель на соответствующий список помещается в нужное поле bif-узла.

Функция read_ll_nodes()

Узлы нижнего уровня в dep-файле имеют разную длину. Эти узлы обычно занимают большую часть dep-файла. Они используются для представления выражений. Действительно, ll-узел, отображающий самоопределенную константу целого типа, значительно отличается от узла, отображающего бинарную операцию. Структура, для ll-узлов dep-файла отражает только те поля, которые присутствуют во всех ll-узлах.

```
struct ll_nd
{ // структура для ll-узла dep-файла
   u_short id;
   u_short variant;
   u_short type;
};
```

После чтения bif-узлов указатель dep-файла должен находиться у начала области ll-узлов. Осуществляется проверка – так ли это. Если не так, указатель файла устанавливается принудительно на начало соответствующей области. Количество ll-узлов известно. Все ll-узлы SAGE-графа (пока почти пустые) на месте. Указатель на первый узел имеется. В цикле из dep-файла считывается структура приведенного выше типа в переменную такого же типа. Значения полей этой переменной, так, или иначе, переносятся в соответствующие поля структуры llnd SAGE-графа. Теперь, поскольку известно значение поля `variant`, то известно, какую сущность представляет узел и, следовательно, какие еще поля следуют за уже прочитанными. После считывания головной части узла указатель dep-файла находится в нужном положении.

read_symb_nodes()

Узлы символов в dep-файле, так же, как и узлы нижнего уровня, имеют разную длину. Структуры символов dep-файла отображают лишь те поля, которые присутствуют во всех узлах символов. Ниже приведена эта структура:

```
struct sym_nd
{ // структрура для symbol-узла dep-файла
   u_short id;      // идентификатор (номер) символа
   u_short variant; // вид символа
   u_short type;    // ссылка на тип данных
   u_short attr;    // ссылка на атрибут
   u_short next;    // ссылка на следующий символ
   u_short scope;   // область видимости
   u_short ident;   // ссылка на символьную строку (имя)
};
```

За этой структурой в dep-файле могут следовать другие поля, зависящие от значения поля **`variant.`**

read_type_nodes()

Узлы типов в dep-файле, так же, как и узлы нижнего уровня, имеют разную длину. Структуры типов dep-файла отображают лишь те поля, которые присутствуют во всех узлах типов. Ниже приведена эта структура:

```
struct typ_nd
{
   u_short id;
   u_short variant;
   u_short name
};
```

За этой структурой в dep-файле могут следовать другие поля, зависящие от значения поля `variant.`

Структуры, представляющие метки, комментарии и имена файлов, имеют простой вид и переносятся с помощью функций `read_label_nodes(), read_cmnt_nodes(), read_filename_nodes().`

По завершении работы конструктора класса SgProject можно использовать остальные классы пакета.

3. Адаптация SAGE - пакета для ОС Windows 2000

Цель адаптации SAGE-пакета заключалась в том, чтобы предоставить пользователю возможность работать на персональном компьютере под управлением операционной системы Windows 2000, используя привычный оконный графический интерфейс. Для адаптации использовалась инструментальная среда C++Builder. В процессе адаптации был выявлен и устранен ряд препятствий, обусловленных тем,

что SAGE-пакет изначально разрабатывался для использования в пакетном режиме под управлением ОС UNIX.

Основные направления адаптации SAGE-пакета:

1. Расширен набор понимаемых парсером расширений для ФОРТРАН - программ (`*.f, *.for, …`).
2. Явно указан режим открытия dep - файла как двоичный. По умолчанию файл открывался как текстовый.
3. Изменен ряд функций из библиотеки SAGE пакета:
   - В функцию `SgOpenProject()` введена инициализация переменных. В исходном варианте пакета SAGE инициализации глобальных и статических переменных не производилось, поскольку предполагалось, что работа с пакетом будет всегда начинаться с “чистого листа”, а завершаться - функцией `exit()`. В этом случае соответствующие переменные изначально обнулены. Для интерактивной работы такое решение неприемлемо. При любом исходе сеанса работы задача (в смысле Windows) не должна завершаться, и не должна терять связь с пользователем.
   - Введен учет использования динамической памяти. Во многих функциях пакета широко используется динамическая память, выделяемая функциями `malloc(), calloc(), new`. Эта память в большинстве случаев не возвращалась, т.к. подразумевалось, что при завершении программы возврат памяти выполнит операционная система. Кроме того, в некоторых случаях динамическая память выделялась как массивная переменная по `new`, а возвращалась (не всегда) как простая переменная, что является ошибкой. Теперь вся выделенная память при завершении сеанса возвращается.
   - Заменен поток вывода диагностических сообщений и сообщений об ошибках: вместо стандартного потока вывода (`stderr, stdout`) организован вывод в специальное окно пользователя.
4. Для работы с большими программами создан дополнительный инструмент, облегчающий разделение программы на блоки определенного размера и вставку include-файлов.
5. Парсер из консольной программы был превращен в интерактивную программу с графическим интерфейсом. Это позволило многократно запускать парсер, выбирая в диалоге различные исходные текстовые файлы программ и задавая различные опции. При каждом запуске парсера диагностические сообщения выдаются в отдельную область диалогового окна.
6. Исправлен метод label() класса SgArithIfStmt, который возвращал неправильную ссылку на метку.

Программы SAGE - пакета, подвергшиеся изменению:

annotate.tab.c, low_level.c, toolsann.c, unparse.c, db.c, list_1.c, setutils.c, writenodes.c, libSage++.c.

4. Использование адаптированного SAGE – пакета для анализа программ

Адаптированный пакет SAGE был использован при разработке первой версии системы статического анализа последовательных программ на языке ФОРТРАН, называемой ниже анализатором. В настоящее время анализатор позволяет

1. строить обобщенный граф различных зависимостей (зависимости по управлению, информационные зависимости) для полной программы, состоящей из нескольких программных единиц, или для отдельной программной единицы;
2. выполнять некоторые виды анализа этого графа;
3. осуществлять визуализацию полученной информации.

При работе с парсером было обнаружено, что в предложении IMPLICIT тип REAL*8 воспринимается как REAL. (Тип DOUBLE PRECISION воспринимается правильно.) Кроме того, парсер накладывает ограничение на размер программ. Большие программы приходится разбивать на отдельные блоки, не превышающие 5000 строк. Парсер обрабатывает предложения INCLUDE, но на практике оказалось удобнее вставлять соответствующие тексты в тело основной программы до ее разделения на части приемлемого размера.

4.1. Анализатор

Исходными данными для анализатора являются:

1. Файл описания проекта *.prj, который представляет собою обычный текстовый файл и содержит список имен dep - файлов. Каждое имя dep - файла располагается в отдельной строке и завершается символами 0x0D, 0x0A.
2. dep - файлы, перечисленные в файле проекта.
3. ФОРТРАН - программы, на основании которых были сформированы dep - файлы.

Комплект этих файлов должен располагаться в одном каталоге.

При использовании SAGE – классов в программе анализа следует иметь в виду, что их методы в каждый момент времени “видят” только один открытый dep - файл проекта. Поэтому при переходе от анализа одной программной единицы, лежащей в одном dep – файле, к анализу другой единицы, лежащей в другом dep – файле, надо своевременно открывать соответствующий dep – файл.

Открытие файла проекта выполняется с помощью функций SAGE-библиотеки и заключается в последовательном открытии заданных в файле проекта dep - файлов. Для каждого dep - файла в памяти создаются структуры, являющиеся внутренним представлением анализируемой программы и отражающие последовательность работы программы и используемые в ней данные.

Дополнительно к SAGE-структурам были созданы другие структуры данных. Одни из них способствуют интеграции программных единиц, разбросанных по разным dep – файлам в единую программу. Другие - представляют информацию об используемых переменных и именованных константах в более интегрированном виде, чем соответствующие структуры SAGE. На основании этих структур выполняется построение объектов, которые позволяют "видеть" и анализировать всю программу в целом, так и отдельные ее компоненты.

4.2. Структуры анализатора

Структуры анализатора характеризуют:

- проект в целом,
- отельный файл проекта
- отдельную программную единицу.

Ниже кратко описаны эти структуры.

4.2.1. Список дескрипторов файлов проекта

Дескрипторы файлов характеризуют отдельные файлы проекта и имеют структуру следующего вида:

```
struct TProjPartInfo_S {
   AnsiString FileName; //имя файла в проекте
   int BaseLineNo;      //номер файла в проекте
   T_CommGN *CommGrNo;  //расширенный дескриптор
};
```

Расширенный дескриптор является объектом класса, в котором свойство cfsmap представляет собой граф потока управления для совокупности программных единиц, входящих в файл. Граф реализован в виде ассоциативного массива. Каждый элемент массива (узел графа) соответствует одному предложению исходного текста (за исключением команд ввода/вывода, команд форматирования, комментариев).

```
class T_CommGN
{
      . . . . . .
   map <int,T_GraphNode> Cfsmap; // обобщенный граф для i-го файла
      . . . . . .                // проекта (ассоциативный массив)
};
```

Ключом в этом массиве является уникальный (в пределах файла) идентификатор, полученный из dep-дерева и строго соответствующий исходному предложению. (Конструкция логический `IF(<...>)` **<...>** может размещаться на одной строке, но в ней использованы два разных идентификатора). С каждым ключом связан блок данных, содержащий характеристики узла, из которых основными являются: описание переменных в левой и правой частях предложения, списки ссылок на

точки программы откуда приходит управление и куда передается управление (списки смежных вершин), имя вызываемой программы или функции и аргументы вызова.

4.2.2. Программные единицы проекта

Совокупность программных единиц проекта (PROGRAM, SUBROUTINE, FUNCTION) представлена ассоциативным массивом, в котором ключом служит имя программной единицы, а значение содержит информацию о программной единице. Во-первых, это информация, позволяющая однозначно идентифицировать программную единицу: номер dep – файла в проекте, номер программной единицы в dep – файле, идентификатор предложения – заголовка программной единицы, имя программной единицы и ее разновидность. Во-вторых, это – перечни объявленных в ней объектов: список формальных параметров, список именованных констант и их значения, список COMMON-блоков со списками входящих в них COMMON-переменных, список эквивалентностей, перечень всех объявленных имен переменных с их характеристиками (локальная, глобальная, формальный параметр, и т.п.).

4.2.3. COMMON-переменные

Все COMMON-переменные, объявленные в программных единицах проекта, отображаются ассоциативным массивом, в котором ключом является имя COMMON-блока, а значением – список дескрипторов тех программных единиц, в которых описан соответствующий COMMON-блок. Здесь дескриптор содержит информацию, схожую с описанной выше. Более того, списки COMMON-переменных, списки эквивалентностей и списки формальных параметров хранятся в одном экземпляре, а дескрипторы содержат ссылки на них.

4.3. Функции анализатора

Функции собирают сведения по всему проекту в целом и сохраняют их до закрытия проекта. Функции выполняются по каждому запросу пользователя. Ниже приведен список и краткое описание этих функций:

Показать для выбранной глобальной переменной все строки, где переменная определяется (DEF). (Массивы в первой реализации рассматриваются как простые переменные.) Выполняется проход по всем процедурам проекта и поиск предложений присваивания. Имя переменной из левой части предложения сравнивается с именем заданной переменной. При сравнении учитываются синонимы, возникшие из-за предложений EQUVALENCE, COMMON, или из-за передачи параметров при вызовах процедур. Пока учитываются только предложения присваивания. В дальнейшем будут учтены и другие предложения (DO, READ), в которых также происходит определение переменных.

Показать для выбранной глобальной переменной все строки, где переменная используется (USE). Работает аналогично предыдущему пункту, только сравнение производится для имен переменных из правой части предложений присваивания.

Показать цепочку вызовов процедур, начиная с заданной процедуры. Цепочка представляет собой иерархическое дерево вызовов, организованное в памяти в виде списка списков. Элемент списка содержит имя процедуры и, при необходимости, аргументы вызова. Вершина дерева - первый (и единственный) элемент первого списка - процедура, от которой начинается сбор цепочки. Каждый список соответствует одному уровню иерархии. В списке одного уровня для каждой процедуры имеется не более одного элемента. При сборе можно исключить из рассмотрения (отключить из зоны видимости) все библиотечные подпрограммы и/или процедуры, указанные пользователем. Можно задать процедуры, которые надо отображать вместе с фактическими параметрами вызова. В последнем случае количество элементов (процедур) в списках одного уровня иерархии может увеличиться (несколько вызовов одной процедуры с разными фактическими параметрами). Цепочка вызовов отображается в виде иерархического дерева (TreeView), которое может быть представлено на экране в виде графа, а также выдано на печать в виде иерархического списка и/или в виде графа.

Собрать все пути для заданной программной единицы и построить по ним матрицу достижимости. Пути собираются в ассоциативном массиве, в котором ключом является порядковый номер пути, формируемый в процессе сбора пути, а элементом массива является также ассоциативный массив. Ключом массива "нижнего уровня" является уникальный идентификатор узла (id), а значением - структура, одно из полей которой содержит номер строки исходного текста, связанной с данным узлом. Матрица достижимости представляет собой ассоциативный массив, число элементов которого равно числу узлов в графе данной процедуры. Эти элементы образуют строки матрицы. Элементами строк являются также ассоциативные массивы, образующие колонки матрицы. Ключами всех массивов служат уникальные идентификаторы узлов (id). Выбор такой структуры для отображения матрицы определялся, во-первых, тем, что эту структуру необходимо формировать динамически, а определение двумерного массива в языке C++ (в Borland CBuilder) требует статического задания второго индекса, и во-вторых, тем, что каждый элемент массива как-то должен быть связан с узлом, а порядковый номер элемента массива, как правило, не соответствует идентификатору узла.

Показать все программные единицы, используемые в данном проекте. Функция перечисляет все программные единицы вместе с формальными параметрами.

Показать все переменные, объявленные в данной программной единице. Функция перечисляет имена всех переменных, объявленных в данной программной

единице, вместе с признаками: глобальная, локальная, эквивалентная, формальный параметр.

Показать все COMMON-блоки, объявленные в данном проекте. Функция перечисляет имена всех COMMON-блоков, объявленных в данном проекте.

Показать COMMON-блоки, объявленные в заданной программной единице. Функция предъявляет все COMMON-блоки, объявленные в данной программной единице, вместе с перечнем принадлежащих им переменных.

Показать синонимы для указанной переменной (в конкретной программной единице). Функция предъявляет все синонимы данной переменной, возникшие в результате использования предложений **EQUIVALENCE** и использования разных имен для переменных, занимающих одинаковые позиции в одноименных **COMMON**-блоках, объявленных в разных программных единицах.

Показать связь формальных и фактических параметров для заданной программной единицы. Функция для каждого формального параметра предъявляет список пар ***<имя1><имя2>***, где ***имя1*** – имя программной единицы, содержащей вызов данной программной единицы, а ***имя2*** – имя переменной, использованной в качестве фактического параметра вызова. Списки строятся только для тех фактических параметров, которые представляют собою ссылки на простые переменные.

4.4. Использование графического пакета GraphViz

Кажется привлекательным располагать возможностью представлять некоторую информацию об изучаемой программе в графическом виде. Для этой цели мы пытались приспособить программные продукты DOT, TWOPI и NEARTO созданные в компании ATT [6,7]. Эти три программы используют один входной интерфейс, но рисуют графы по-разному. Программы запускаются из командной строки, в которой задается имя входного файла и совокупность управляющих опций. Входной файл – текстовый, и в нем описано множество связанных пар объектов в форме ***имя узла -> имя узла.*** Для каждой пары программа рисует два узла, имеющих форму некоторой геометрической фигуры (по умолчанию - эллипс), помещает внутри каждой фигуры имя узла и соединяет узлы линией со стрелкой (ребро графа).

Программа DOT работает с направленными графами, и ее алгоритм размещения узлов основан на понятии “ранг узла”. Ранг это – численная характеристика узла. Ранг узла, находящегося слева от стрелки меньше ранга узла, находящегося справа от стрелки. Ранги узлов вычисляются последовательно по мере сканирования входного файла. Узлы одного ранга размещаются на одном (по умолчанию горизонтальном) уровне. Чем больше ранг, тем (по умолчанию) ниже уровень. В пределах одного уровня места узлов подбираются так, чтобы обеспечить минимально возможное число пересечений ребер получаемого графа и минимальную

их длину. Имеется много опций для управления видом получаемого графа. Опции выбора формы, цвета контура и цвета заливки узлов. Опции вида и цвета линий для ребер графа. Опции, позволяющие помещать произвольные надписи на узлах и ребрах. Опции, позволяющие задавать позиционирование уровней рангов снизу вверх, сверху вниз, слева направо или справа налево. Опции, управляющие формой и положением стрелок на концах ребер графа. И много других.

Программа TWOPI работает во многом похожим образом, но размещает узлы графа на концентрических окружностях, а не на прямых линиях. При этом узлы с большим рангом размещаются на окружностях большего радиуса.

Программа NEARTO работает с ненаправленными графами и использует иной алгоритм размещения узлов графа. Алгоритм предполагает, что ребра графа обладают упругостью, и узлы графа располагаются так, чтобы “энергия” системы была минимальной.

Мы встроили программу DOT в нашу программу анализа для наглядной демонстрации цепочки вызовов программных единиц. Пример полученного графа приведен в Приложении 1.

5. Заключение

Наш ограниченный опыт работы с пакетом SAGE показал пригодность использования пакета для анализа ФОРТРАН - программ в условиях интерактивной среды, при некоторой его доработке. Дополнительные доработки улучшили бы его функциональность:

- Попытаться преодолеть ограничения, которые парсер накладывает на размер исходных программ. Возможно, эти ограничения являются следствием использования компилятора компиляторов Bison, в котором сохранились следы от его предназначения для старых 16-разрядных вычислительных машин. Для этого надо Bison погрузить в современную инструментальную среду и изучить его исходные тексты.
- Попытаться изменить парсер так, чтобы он для предложений `END, ENDIF, ENDDO` и `THEN` порождал бы соответствующие узлы dep-дерева – для каждого предложения свой узел, а не один `CONTROL_END` для всех, как сейчас. Это упростило бы разработку программ анализа.
- Дописать методы классов SAGE, которые объявлены, но не реализованы.
- Перевести на русский язык сообщения, выдаваемые компонентами пакета SAGE пользователю.
- Переписать некоторые компоненты SAGE, используя современный стиль программирования на языке C++ с целью придания пакету большей стройности.

- В случае использования пакета для работы только с языком ФОРТРАН, удалить все, что связано с языком Си. Размер пакета значительно уменьшится, его структура будет намного яснее, а работа надежнее.

Сейчас на вход парсера подается текст на языке ФОРТРАН, возможно содержащий ошибки. Парсер не все ошибки обрабатывает корректно. Правильно было бы предварительно пропускать входной текст через специальную программу, осуществляющую синтаксический контроль входного текста (ФОРТРАН - компилятор), и только после этого передавать текст на вход парсера.

Авторы выражают благодарность К.Н. Ефимкину за многие полезные обсуждения и поддержку данной работы.

Литература

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления – СПб.: БХВ-Петербург, 2002.
2. www.parallelsp.com
3. www.nas.nasa.gov/tools/CAPO
4. C. Ierotheou, S.P. Johnson, M. Cross, P.F. Leggett, “Computer Aided Parallelisation Tools (CAPTools) – conceptual overview and performance on the parallelisation of structured mesh codes”, *Parallel Computing*, Vol. 22, pp 163-195, 1996.
5. www.extreme.indiana.edu
6. www.graphviz.org
7. www.research.att.com/sw/tools/graphviz/download.html

Приложение 1. Пример представления в графической форме цепочки вызовов процедур/функций для реального большого приложения